PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46

A1

(11) International Publication Number: **WO 98/03910**

(43) International Publication Date: 29 January 1998 (29.01.98)

(21) International Application Number: PCT/GB97/02006

(22) International Filing Date: 24 July 1997 (24.07.97)

(30) Priority Data:

9615532.0	24 July 1996 (24.07.96)	GB
9621947.2	22 October 1996 (22.10.96)	GB

(81) Designated States: JP, US, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

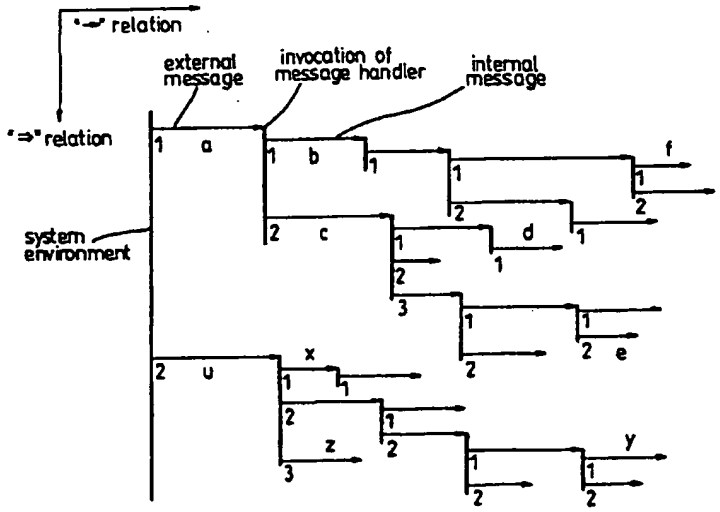
Published
With international search report.

(71) Applicant *(for all designated States except US)*: HEWLETT-PACKARD COMPANY [US/US]; 3000 Hanover Street, Palo Alto, CA 94304 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: CARROLL, Jeremy, John [GB/IT]; Via Ernesto Rossi, 65, I-57125 Livorno (IT). BORSHCHEV, Andrei Vladilenovich [RU/RU]; Socialisticheskaya, 4-33, St.Petersburg, 191002 (RU).

(74) Agents: LAINE, Simon, James et al.; Wynne-Jones, Laine & James, 22 Rodney Road, Cheltenham, Gloucestershire GL50 1JJ (GB).

(54) Title: ORDERED MESSAGE RECEPTION IN A DISTRIBUTED DATA PROCESSING SYSTEM



(57) Abstract

A complex computing system has a plurality of nodes interconnected by channels through which data messages are exchanged. The underlying principle is that after arrival at a node of a message, delivery of that message is delayed until after delivery and consequences of all more senior messages which affect the node. The messages are progressively timestamped at each node so that each time stamp contains generation by generation indicators of the origin of the associated message. The seniority of that message is uniquely determined thereby and total ordering of the messages can be achieved. When comparing timestamps for such ordering, comparison of respective generation indicators is necessary only until there is a distinction.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

ORDERED MESSAGE RECEPTION IN A DISTRIBUTED DATA PROCESSING SYSTEM

This invention relates to complex computing systems. It was developed primarily to answer a problem with distributed systems, but it has been realised that it is equally applicable to systems which, are not normally considered to be distributed, such as a multi-processor computer. Although their physical separation may be negligible, nonetheless the processors are distinct and form a "distributed" system within the computer to which this invention is applicable.

A landmark paper on distributed systems is that of Lamport ("Time, Clocks and the Ordering of Events in a Distributed System" - Communications of the ACM Vol. 21 No. 7, 1978 pp 558-565). In that, a distributed system is defined as a collection of distinct processes which are spatially separated and which communicate with one another by exchanging messages, and in which the message transmission delay is not negligible compared to the time between events in a single process. In such a system, it is sometimes impossible to say that one of two events occurred first. Lamport proposed a logical clock to achieve a partial ordering of all the events, and he postulated a single integer timestamp on each message, corresponding to the time the message was sent.

Fidge (in "Logical Time in Distributed Computing Systems" - IEEE Computer 24(8) August 1991 pp 28-33) argued that the time stamps of Lamport clocks (totally ordered

logical clocks) impose on unrelated concurrent events an arbitrary ordering, so that the observer cannot distinguish from genuine causal relationships. He proposed partially ordered time readings and timestamping rules which enable a causal relationship between two events to be established. Their order could then be determined. But where there is no causal relationship between events, no definitive order exists, and different total orderings of events (or interleavings) are possible. This means that some messages are assigned an arbitrary order.

This ordering problem is known as the "race condition problem" and it can be illustrated by a simple analogy. A dictates a first message to secretary B, who faxes the typed version to C. A telephones C with a second message. Unless ordered, the communication system will not know whether the first or second message reached C first, although it will know that the dictation preceded the fax.

It is the aim of this invention to resolve this problem and to allow someone to programme a distributed system as if he was programming a uni-processor. In other words, he can think about time linearly and he will not have to be concerned about concurrency or the race condition problem.

According to one aspect of the present invention there is provided a complex computing system comprising a plurality of nodes connected to each other by channels along which timestamped data messages are sent and received, each timestamp being indicative, generation by generation, of its seniority acquired through its ancestors' arrival in the

system and in any upstream nodes, and each node comprising:
means for storing each input data message,
means for determining the seniority of input data messages by progressive comparison of respective generations in the timestamps until the first distinction exists,
means for delivering these messages for processing,
means for applying a timestamp to each output message derived from such processing comprising the immediately ancestral message's timestamp augmented by a new generation seniority indicator consistent with the ordering, and
means for outputting such ordered and timestamped messages.

The delivery means will generally be arranged to deliver messages in order according to which message has the most senior timestamp indicator.

For a data message received from outside the system the initial timestamp indicator will preferably include an indication of the time of receipt of said data message at the node, while for a data message generated by a node of the system the new generation seniority indicator of the timestamp will preferably include an indication of the place of said data message in the ordered sequence of such messages at said node. This indication may be real time or logical time.

Conveniently, monotonic integers are utilised as said generation seniority indicators in the timestamps.

Advantageously, the delivery means of a node delivers data messages only either once a message has been received on each of the input channels of said node or when at least

one data message received on each of the input channels of said node is stored in the storage means.

Preferably each node will be adapted to perform at least one channel flushing routine triggerable by lack or paucity of channel traffic.

Ideally, all data messages caused by a first data message anywhere in the system will be delivered to a node before any messages caused by a second data message, junior to the first data message, are delivered to said node.

According to another aspect of the present invention there is provided a method of ordering data messages within a complex computing system comprising a plurality of nodes connected to each other by channels along which data messages are sent and received, the method comprising, for each node, timestamping each message on arrival, queuing messages until a message has been received on each input channel to the node, and delivering the queued messages for processing sequentially in accordance with their timestamps, the message having the most senior timestamp being delivered first, wherein the timestamping at each node is cumulative so that the timestamp of a particular message indicates the seniority acquired by that message, generation by generation, and wherein the seniority of one message against another is determined by the progressive comparison of respective generations in the timestamps until the first distinction exists.

According to a further aspect of the present invention there is provided a complex computing system comprising a

plurality of nodes between which data messages are exchanged, wherein after the arrival at a node of a message, delivery of the message by the node is delayed until after the delivery and consequences of all more senior messages which affect the node.

Such a system may be either a distributed computing system, a symmetric multi-processor computer, or a massively parallel processor computer.

Assumptions

To understand later explanations, certain assumptions about a distributed computing system will be set out.

Such a system is a set of nodes or processes connected by FIFO (first in, first out) channels. Conventionally, 'nodes' refer to the hardware and 'processes' to the software and operations performed at the nodes, but the terms may be used interchangeably here. Some of these processes have external channels through which they communicate with the system's environment, the whole system being driven by input messages through some external channels, and sending out an arbitrary number of consequential output messages through other external channels.

Each process can be regarded as an application layer and a presentation layer, which handle the following events:

(a) Message arrival (at the presentation layer)

(b) Message delivery (from presentation to application layer)

(c) Message send request (from application to presentation layer)

(d) Message send (from presentation layer)

(e) Message processing complete (from application to presentation layer).

At any event (b), the application layer

i) generates one or more events (c)

ii) changes the process state, and

iii) generates event (e) - which indicates that it is ready to receive a further message.

A set of such events will be termed a message handler invocation. Such invocations are the basic building blocks or atomic units of a distributed system, and a process history is a sequence of such invocations. Each invocation may affect subsequent invocations by changing the internal state of the process.

At the application layer, the channels are simplex. However, auxiliary messages, from one presentation layer to the other, are allowed in both directions.

There will be a global real-time clock, accessible from anywhere in the system. It is required only to be locally monotonic increasing, and there will be some bound on the difference between two simultaneous readings on the clock in different processes.

The FIFO channels are static.

Processes do not generate messages spontaneously.

Each message in the system has exactly one destination.

(These last three assumptions are working hypotheses which will be relaxed later).

Finally, for initial consideration, there are no loops

in the possible dataflows. This will be discussed further below.

The time model

The aim is to achieve a total ordering of the set of messages in the system. If messages are delivered to every process in time order and messages are sent along every channel in time order then the system is said to obey "the time model". A total ordering of the set of messages is equivalent to an injective mapping from the set of messages in the system to a totally ordered set (the time-line).

In this specification "<" will signify, in the relationship $m_p < m_q$, that message m_p precedes message m_q in the total order. This gives the first principle of the time model: *there is a unique time for everything*. Such a time is simply a label useful for evaluating the time order relationship between messages, and does not have any necessary relationship with real time.

The relation "<" is based on two partial order relations, "$\Rightarrow$" (sent before) and "$\rightarrow$" (strong causality), as explained below.

For any two external input messages, m_0 and m_1, either $m_0 \Rightarrow m_1$ or $m_1 \Rightarrow m_0$. This is given by the environment, typically by the clock time of message arrival. In other words, external input messages are totally ordered with respect to $\Rightarrow$.

If message send requests for messages m_0 and m_1 occur during the same invocation at the behest of some third message, the send request for m_0 being before m_1, then $m_0 \Rightarrow$

m_1.

"→" is the least partial order such that if the message send request for m_1 occurs during the invocation in response to m_0, then $m_0 \rightarrow m_1$. (i.e. a message strongly causes any messages sent by its handler).

The total order relation "<" is determined by the following axioms:

If $m_0 \Rightarrow m_1$ then $m_0 < m_1$.

If $m_0 \rightarrow m_1$ then $m_0 < m_1$.

If $m_0 \Rightarrow m_1$, $m_0 \rightarrow m'_0$ then $m'_0 < m_1$.

The first two axioms correspond to Lamport's axioms; the third, the strong causality axiom, is the heart of the time model of the present proposal.

The idea behind the strong causality axiom is the following: if a process or the system's environment sends two messages (m_0 and m_1), one after another, then any consequence (m'_0) of the first message (m_0) should happen before the second message (m_1) and any of its consequences.

This gives the second principle of the time model: *there is enough time for everything* (i.e. enough time for all remote consequences to happen before the next local event).

For a better understanding of the invention reference will now be made, by way of example, with reference to the accompanying drawings, in which:

Figure 1 is a diagram illustrating the total ordering of messages,

Figure 2 is a diagram of a process with its time

service,

Figure 3 illustrates a message sequence of the time service,

Figure 4 shows a network of processes, to explain channel flushing,

Figure 5 is a diagram showing channel flushing messages and the structure of time service,

Figure 6 illustrates a message sequence of channel flushing,

Figure 7 shows a bus,

Figure 8 shows a delay,

Figure 9 shows a false loop, and

Figure 10 comprises diagrams of feedback through a bus.

Referring to Figure 1, the diagram can be likened to a tree on its side with its root (to the left) representing the system's environment which generates external messages. The nodes (the vertical lines) are message handler invocations and the arrowed horizontal lines represent messages.

Using this tree it is easy to reconstruct the message relations. For example, $a \rightarrow b$ because b was sent while a was handled; $b \Rightarrow c$ because b was sent before c in the same invocation. Also, $a \rightarrow f$ and $x \Rightarrow z$ as these relations are transitive. f and e are incomparable under both "$\Rightarrow$" and "$\rightarrow$"; nevertheless $f < e$. To compare two messages with respect to the total order relation "$<$" one has to trace paths from the root to these messages. There can be three possible cases, which correspond to the three axioms. They are shown by the

following three examples taken from Figure 1:-

c lies on the path of d, hence, $c \rightarrow d$ and therefore $c < d$.

x and z have the same path, but x is sent before z, so $x \Rightarrow z$ and $x < z$.

f and e have the same path prefix, but then their paths fork, and b (with $b \rightarrow f$) is sent before c (with $c \rightarrow e$), which means $b \Rightarrow c$, so $f < c < e$, giving $f < e$.

If a distributed system follows this time model, i.e. if messages are delivered to each process in this order, and sent down each channel in this order, then the system's behaviour will be deterministic, independent of the speed of processes and channels.

A possible time-line from which each message can be given a unique time is the set of sequences of integers. These are ordered using the standard dictionary ordering.

The path from the root to a message fully identifies the message position in the total order relation "<". This path can be codified as a representation of time in the distributed system. The names of processes along the path are immaterial; the only information needed is the relative order of the message ancestors at each process and the order of the initial external messages. So, in Figure 1, the time can be represented by an array of integers, e.g. [1,2,3,1,2] for e, [2,3] for z, [2,2,2,1,1] for y. However, since the system's environment may be distributed, it could be difficult to assign unique integers to each external message. A possible solution is to use real clock values

combined with an external input identifier, this requiring that at every external input all real clock readings are unique and grow monotonically.

The following C++ class can be used for time:

```
class TTime {
public:
    TTime():
        RealClock( 0.0 ),
        Input( 0 ),
        Length(0) {}
    TTime( float realclock, unsigned input ):
        RealClock( realclock ),
        Input( input ),
        Length(0) {}
    void AddNewProcess()
        { Path[ Length++ ] = 0; }
    void operator++() { Path[Length-1]++; }
    friend bool operator<( TTime t0, TTime t1 );
private:
    float RealClock;
    unsigned Input,
    unsigned Path[ MAX_PATH ];
    unsigned Length;
};
```

```
friend TTime bool operator<( TTime t0, TTime t1 ) {
    if( t0.RealClock < t1.RealClock ) return TRUE;
    if( t1.RealClock < t0.RealClock ) return FALSE;
    if( t0.Input < t1.Input ) return TRUE;
    if( t1.Input < t0.Input ) return FALSE;
    for( unsigned i=0; i<min( t0.Length, t1.Length ); i++ ) {
        if( t0.Path[i] < t1.Path[i] ) return TRUE;
        if( t1.Path[i] < t0.Path[i] ) return FALSE;
    }
    return t0.Length < t1.Length;
}
```

The more processes handle a message, the longer its path grows. Potentially, if there is a cycle in the system, paths can become arbitrarily long.

The implementation should use (when possible) dynamic allocation to avoid the arbitrary upper limit on path length.

Each node or message handler invocation in the distributed system is structured as shown in Figure 2. All functionality related to support for the time model resides in the time service, so that a process does not know

anything about the time model. Each time it finishes handling a message it informs its time service (*Done* signal or event (*e*) above). The time service has a local clock *T* of type *TTime* which is updated whenever a message is received or sent by its process. Initially the local clock has a value given by a default constructor and it is kept and used even while the process is idle.

The "timestamp assigner" at the border with the system's environment has a real clock synchronized with the clocks of all other timestamp assigners, and a unique input identifier. 'Synchronized' here is understood to mean adequately synchronized, for example by means of the network time protocol as described in the Internet Engineering Task Force's Network Working Group's Request for Comments 1305 entitled 'Network Time Protocol (Version 3) Specification, Implementation and Analysis' by David L Mills of the Univ. of Delaware published by the IETF in March 1992. Each time an external message enters the system it gets a unique timestamp constructed from these two values (see the second constructor of *TTime*). It is assumed that the real clock progresses between each two messages.

Input messages are not delivered to the process until there are messages present on all inputs. Once this condition holds, the local clock is set to the timestamp of the most senior message, the new process is added to the path, and the most senior message is delivered. Every output message sent by the process while handling this input is timestamped by the time service with the current value of

the local clock; and then the clock is incremented. The next message can be delivered only after the process explicitly notifies the time service that it has finished with the previous one, is idle and waiting (*Done*). The corresponding message sequence is shown in Figure 3. The basic algorithm Alg.1 of the time service is shown in the table below.

Initial state: Idle.	
Event	Action
State Idle	
Input message arrives	if(there are messages on all inputs) *// If all inputs are non-empty,* DeliverTheOldestMessage(); *// delivery is possible*
State Handling Message	
Process sends output message	Send it with the timestamp T; T++; //increment the last time in the timestamp
Done	if(there are messages on all inputs) { *// If all inputs are still non-empty,* DeliverTheOldestMessage(); *// deliver the next oldest message* return; } Next state = Idle;
Functions	
void DeliverTheOldestMessage() { T = timestamp of the oldest message; *// First, the local clock is set to the value of the oldest* T.AddNewProcess(); *// timestamp, and a new process is added to the path in it.* Deliver(the oldest message); Next state = Handling Message; }	

This algorithm ensures that input messages are delivered to each process in the order of their timestamps, and that output messages are sent by each process in the order of their timestamps. Thus, the time service as described above fully implements the time model.

However, a distributed system containing a cycle will not work, as all time services in the cycle will always be

missing at least one input message. Also, rare messages, either on an external input or on an internal process-to-process connection, may significantly slow down the whole system.

Channel flushing can solve both these problems. Channel flushing is a mechanism for ensuring that a message can be accepted. The principle is to send auxiliary messages that enable the time service to prove that no message will arrive which is earlier than one awaiting delivery. Hence the waiting message can be delivered.

There are two kinds, namely 'sender channel flushing', in which the sending end initiates channel flushing when the channel has been left unused for too long, and 'receiver channel flushing', in which the receiving end initiates channel flushing when it has an outstanding message that has been awaiting delivery for too long.

Receiver channel flushing will be considered first, in conjunction with Figure 4. For simplicity, timestamps and clocks are represented by single integers.

Suppose for a certain period of time the two lower inputs of the process *C* are empty while there is a message with timestamp 23 waiting on the upper input. The time service of *C* wants to deliver the message as soon as possible, but it cannot do so until it proves that those messages that will eventually arrive on the empty inputs will have greater timestamps.

To prove it, *C* sends a channel flushing request "May I

accept a message of time 23?" to *B* and *F*, both of which have to forward this request deeply into the system, until either a positive or negative response can be given. In fact, to verify that *C* can accept the message with the timestamp 23 in Figure 4, it is enough to ask only the processes shown, since all inputs to the diagram are at times later than 23.

The algorithm described below is a straightforward implementation of channel flushing. All channels in the system (which actually connect the time services of the processes) are bi-directional, since, besides the normal uni-directional messages, the channel flushing messages are sent along them in the reverse direction. These messages and the structure of the time service are shown in Figure 5.

The general idea is that each time the time service discovers that there are input messages waiting while some inputs are empty, it sets a flush timer. On the timeout event it starts the channel flush. It sends flush requests to all empty inputs, creates a (local) request record with the list of these inputs, and then waits for responses. If positive responses come from all inputs, the oldest message is delivered. If a negative response comes on any input the flush is cleared and re-scheduled.

Requests from other time services are handled in the following way. First, the time service tries to reply using its local information (its local clock and the timestamps of waiting messages). If it is unable to do so, it creates a (remote) request record and forwards the request to all

empty inputs. If all responses are positive, so is the one to the remote requester. Otherwise, the response is *No*.

The algorithm Alg.2 is presented in a table below. Again, the time service has two states: *Idle* and *Handling Message*. While in the latter state, the time service is only serving process output messages, whereas in the Idle state it does all channel flushing work for both itself and other processes. < *t, Path, Inputs* > represents a request record. *[]* is an empty path. *New, Delete* and *Find* are operations over an array of request records that maintain the local state of the receiver channel flush algorithm. *P* is the identifier of this process. *T* is the local clock.

Initial state: Idle, Flush timer not set, no request records.	
Event	Action
State Idle	
Input message with time t, arrives at input i	for(all < t, Path, Inputs >): Path ≠ []) *// First update remote requests.* if(t < t_i) *// Input i is younger than the request's time.* YesForRequest (< t, Path, Inputs >, i); else *// Input i is older than the request's time.* NoForRequest (< t, Path, Inputs >); if(there are messages on all inputs) { *// Then, if all inputs are non-empty,* CancelLocalFlushing(); DeliverTheOldestMessage(); *// delivery is possible.* return; } if(the new message is the oldest one) { *// If this message becomes the* CancelLocalFlushing(); *// oldest one, a new local* Set flush timer; *// flushing must be scheduled.* return; } if(Find(< t, [], Inputs >)) *// Otherwise, if there is a local request waiting* YesForRequest (< t, [], Inputs >, i); *// for this input, that means Yes.*
Flush timeout	StartFlushing(timestamp of the oldest message, []); *// Empty return path indicates that the request is local.*
< Your Next Time?, t, [P_0...P_n] > *flush request*	if(P is among [P_0...P_n] || t < T) { *// If request has made a cycle – assume Yes.* Send to output to P_n: < Yes, t, [P_0...P_n] >; *// Or, if local clock is already* return; *// ahead of t, definitely Yes.* } if(there is a message older than t on some input) { Send to P_n: < No, t, [P_0...P_n] >; *// Then assume No.* return; } *// This process is not able to answer immediately and it starts flushing.* StartFlushing(t, [P_0...P_n]);
< Yes, t, [P_0...P_nP] > *+ve response on input i*	if(Find(< t, [P_0...P_n], Inputs >)) YesForRequest (< t, [P_0...P_n], Inputs >, i)
< No, t, [P_0...P_nP] > *negative response*	if(Find(< t, [P_0...P_n], Inputs >)) NoForRequest (< t, [P_0...P_n], Inputs >)

State Handling message	
Process sends output message	Send it with the timestamp T; T++;
Done	if(there are messages on all inputs) { *// If all inputs are still non-empty,* DeliverTheOldestMessage(); *// deliver the next oldest message.* return; } if(there is a non-empty input) *// Otherwise, if there is a non-empty input,* Set flush timer; *// a new local flushing must be scheduled.* Next state = Idle;

Functions

```
void DeliverTheOldestMessage() {
    T = maximum(T, timestamp of the oldest message ); // Before delivery, the local clock is set
    T.AddNewProcess();                                // to the value of the oldest timestamp, and
    Deliver( the oldest message );                    // a new process is added to the path in this value.
    Next state = Handling Message;
}
```

```
void CancelLocalFlushing() {                  // Cancelling local flushing activity includes
    Cancel flush timer;                       // cancelling the flush timer (just in case it is set)
    if( Find( < t, [], Inputs > ) )           // and deletion of a local request record (if any).
        Delete( < t, [], Inputs > );
}
```

```
void StartFlushing( TTime t, TPath path ) {   // Flushing upon local or remote request starts with
    for( all empty inputs )                   // sending requests to all empty inputs
        Send to input: < Your next time?, t, Path+P >;   // (P is added to the return path)
    New( < t, Path, Set of empty inputs > );  // and creation of a new request record.
}
```

```
void YesForRequest ( TRecord < t, Path, Inputs >, TInput i ) {   // On positive response on input i
    if( i ∉ Inputs )                          // If the record has already received this
        return;                               // information then it doesn't care.
    Inputs = Inputs \ i;                      // The input is removed from the Inputs set of the request record.
    if( Inputs == ∅ ) {                       // If this set becomes empty, no more responses are needed.
        Delete( < t, Path, Inputs > );
        if( Path == [] )                      // and if it is a local request
            DeliverTheOldestMessage();        // the oldest message is delivered.
                // Successful channel flushing.
        else                                  // Otherwise it is a remote request.
            Send to the last process in the Path: < Yes, t, Path >; // Yes is sent to the next process
    }                                         // in the return path.
}
```

```
void NoForRequest ( TRecord < t, Path, Inputs > ) {   // Negative response – acted on immediately.
    Delete( < t, Path, Inputs > );            // The corresponding record is deleted.
    if( Path == [] )                          // If it is a local request
        // Unsuccessful channel flushing
        Set flush timer;                      // then restart the flush timer.
    else                                      // Otherwise, for a remote request.
        Send to the last process in the Path: < No, t, Path >;  // No is sent to the next process
}                                             // in the return path.
```

To illustrate the work of the algorithm, consider the example in Figure 4 in conjunction with one possible channel flushing message sequence as shown in Figure 6. "?" denotes here "Your Next Time?", "Y" stands for Yes, and "N" for No. Sets near the vertical axes represent the sets of processes from which responses are still wanted (Inputs in the above terminology). "ok" means that a process is sure it will not be sending anything older than the timestamp of the request, (i.e. 23). "loop" means that a process has found itself in the return path of the request.

It is evident that the channel flushing procedure consists of two waves: a fanning out request wave and a fanning in response wave. The request wave turns back and becomes the response wave as soon as the information needed for the initial request is found.

The "timestamp assigner" at the border with the environment treats the channel flushing request in the following way.

RealClock() returns the real clock reading; Input is the unique external input identifier.	
Event	Action
External input message arrives	Send it with the timestamp TTime(RealClock(), Input);
< Your Next Time?, t, [P_0...P_n] > *flush request*	if(t < TTime(RealClock(), Input)) *// If t is older than local time* Send < Yes, t, [P_0...P_n] >; *// Then definitely Yes* else Send < No, t, [P_0...P_n] >; *// Otherwise assume No*

Sender channel flushing is conceptually simpler, and significantly more efficient than receiver channel flushing, although it does not provide a solution to the loop problem.

In sender channel flushing, each output channel of each process has a timeout associated with it. This timeout is reset each time a message is sent down the channel. The timeout can be either a logical timeout (i.e. triggered by some incoming message with a sufficiently later timestamp) or a physical timeout. If the timeout expires before being reset then a sender channel flush is initiated down that channel. The channel flush consists of a 'non-message' which is sent down the output channel. The receiver can use it to advance the time of the channel by allowing the time service to accept earlier messages waiting on other channels. When the non-message is the next message to be accepted, then the time service simply discards it. However, the non-message, by advancing the receiver's local clock, can cause logical timeouts on output channels of the receiver; hence causing a cascading sender channel flush.

The timestamp assigners also participate in sender channel flush; they have to use a physical timeout. In general, using both sender and receiver channel flushes is recommended; preferably with some sender channel flushes piggy backed upon receiver channel flush response messages.

To provide usable middleware implementing the time model it is necessary to relax some of the more restrictive assumptions about the system being built. Three special processes that need to be created and integrated with such

middleware are now considered, as is a full treatment of loops in the dataflow.

The bus

The bus is a process that allows multiple output ports from any number of processes to be connected to multiple input ports on other processes. The term 'bus' is taken from Harrison (A Novel Approach to Event Correlation, Hewlett-Packard Laboratories Report No. HPL-94-68 Bristol UK 1994) and is intended to convey the multiple access feature of a hardware bus.

The bus implements a multicast as a sequence of unicasts, its operation being shown in Figure 7.

The output channels are ordered, (shown in the diagram as 1, 2, 3, 4). When a message is delivered by the time service to any of the input channels, the bus outputs an identical message on each of its output channels in order. The time service computes the timestamp for these in the normal way, as shown.

A bus acts as a message sequencer, ensuring that all recipients of a series of multicasts receive the messages in the same order (as shown).

The delay

In a non-distributed system it may be possible to set a timer, and then have an event handler that is invoked when the timer runs out. This alarm can be seen as a spontaneous event. Within the time model, it must be ensured that

spontaneous events have a unique time. The simplest way of achieving this is to treat spontaneous events just like external events. A timestamp is allocated to them using a real clock and a unique input identifier. Moreover, a process can schedule itself a spontaneous event at some future time which again will get a timestamp with real part coming from the scheduled time. Having thus enabled the scheduling of future events the delay component can be created as schematically shown in Figure 8.

For each input message the delay generates an output message at some constant amount of time, δ, later. The time of the generated message is given by the sum of 5 and the first time in the timestamp (the real time part). The rest of the path part of the timestamp is ignored. The input identifier part of the time stamp is changed from the original, 1, to the input identifier of the delay, 1'. There are large efficiency gains from fully integrating delays with the receiver and sender channel flush algorithms. The responses to flush requests should take the length of the delay into account, as should any relaying of flush requests through the delay.

The plumber

The plumber (the topology manager) is a special process that manages the creation and destruction of channels and processes. The plumber has two connections with every process in the system. The first is for the process to make topology change requests to the plumber; the second is for

the plumber to notify the process of topology changes that affect it (i.e. new channels created or old channels deleted). The plumber can create and delete processes that have no channels attached. The plumber has a specific minimum delay between receiving a topology change request and doing it. This is the key to a feasible solution to topology changes within this time model. The reason that topology changes are difficult for (pessimistic implementations of) the proposed time model is that for a process to be able to accept a message it must know that it is the oldest message that will arrive. If the topology is unknown then all other processes within the application must be asked if they might send an older message. This is implausible. The plumber acts as the single point one needs to ask about topology changes. Moreover, the minimum delay between the request for a topology change and its realisation ensures that the plumber does not need to ask backward to all other processes. For large systems, or for fault tolerance, multiple plumbers are needed, and these can be arranged hierarchically or in a peer-to-peer fashion. As with the delay process the plumber needs to be integrated with the channel flush algorithms.

Loops

Loops generate issues for the time model and typical loops generate a need for many auxiliary messages. A loop without a delay can only permit the processing of a single message at any one time anywhere within the loop (it is said

that the loop "locksteps"). Three solutions to these problems are examined.

Removing loops

The traditional design models, client/server, master/slave, encourage a control driven view of a distributed system, which leads to loops. A more data driven view of a system, like the data flow diagrams encouraged in structure analysis, is typically less loopy.

Moreover, where a first cut has loops, a more detailed analysis of a distributed system may show that these loops are spurious. The data-flows, rather than feeding into one another, feed from one submodule to another and then out. For example, in Figure 9, there is an apparent loop between process A and process B, (a flow from A feeds into B which feeds back into A). But when one looks at the sub-processes A1, A2, B1, B2 there are, in fact, no loops, only flows.

Co-locate the processes in a loop

If there is a loop for which other solutions are not appropriate, it will be found that only one process within the loop can be operational at any one time. It will normally be better to have this, and put all the processes in the loop on the same processor. There will be no penalty in terms of loss of parallelism. This approach will minimise the cost of the auxiliary messages, because they will now be local messages.

Break the loop using a delay

Informally, the problem with a loop is feedback. Feedback happens when an input message to a process strongly causes another message (the feedback) to arrive later at the same process. Under the strong causality axiom, feedback is strongly caused by the original messages, and hence comes before all subsequent messages. Hence any process in a loop must, after processing every message, first ascertain whether there is any feedback, before proceeding to deal with any other input. A delay process is a restricted relaxation of strong causality, since each input to the delay does not strongly cause the output, but rather schedules the output to happen later. Hence, if there is a delay within the loop, then a process can know that any feedback will not arrive until after the duration of the delay. Hence it can accept other messages arriving before the feedback.

A difficult case

The example in Figure 10 presents specific problems of both semantics and implementation for feedback.

In each of the four cases we see a message with data α arriving at a bus B and being multicast to processes A and C. A responds to the message α by outputting a message with data β which is fed back into the bus, and hence multicast to A and C. When it arrives at A no further feedback is produced. If the bus sends to C before A (Figure 10a) then no issues arise: the original message is multicast to both parties, and then the feedback happens and is multicast.

If, on the other hand, the bus sends to A before C then the feedback happens before the original message is sent to C. The order in which C sees the feedback and the original message is reversed (Figure 10b). This indicates that strong causality and feedback require a re-entrant processing similar to recursive function calls. Such re-entrant processing breaks the atomicity of invocations and also needs significantly more channel flushing messages than the non-re-entrant algorithm that has been presented. The simplest form algorithm Alg.1 would incorrectly output the later message from B to C before the earlier message (Figure 10c). Without re-entrant processing there is a conflict between strong causality and the sent after relation. The later version algorithm Alg.2, refines the *DeliverTheOldestMessage* function to ensure that all incoming messages are delayed (with the minimum necessary delay) until after the previous output (Figure 10d). This implementation obeys the time model, but (silently) prohibits non-delayed feedbacks. At the theoretical level this obviates the necessity for re-entrancy and prefers the sent after relation to strong causality. At the engineering level, this can be seen as a compromise between the ideal of the time model and channel flushing costs.

A slight more exhaustive account of the above is given in the priority documents accompanying this Application.

CLAIMS

1. A complex computing system comprising a plurality of nodes connected to each other by channels along which timestamped data messages are sent and received, each timestamp being indicative, generation by generation, of its seniority acquired through its ancestors' arrival in the system and in any upstream nodes, and each node comprising:

means for storing each input data message,

means for determining the seniority of input data messages by progressive comparison of respective generations in the timestamps until the first distinction exists,

means for delivering these messages for processing,

means for applying a timestamp to each output message derived from such processing comprising the immediately ancestral message's timestamp augmented by a new generation seniority indicator consistent with the ordering, and

means for outputting such ordered and timestamped messages.

2. A complex computing system as claimed in Claim 1, wherein the delivery means is arranged to deliver messages in order according to which message has the most senior timestamp indicator.

3. A complex computing system as claimed in Claim 1 or 2, wherein for a data message received from outside the system the initial timestamp indicator includes an indication of the time of receipt of said data message at the node.

4. A complex computing system as claimed in Claim 1, 2 or 3, wherein for a data message generated by a node of the system the new generation seniority indicator of the timestamp includes an indication of the place of said data message in the ordered sequence of such messages at said node.

5. A complex computing system as claimed in any preceding claim, wherein monotonic integers are utilised as said generation seniority indicators in the timestamps.

6. A complex computing system as claimed in any preceding claim, wherein the delivery means of a node delivers data messages only once a message has been received on each of the input channels of said node.

7. A complex computing system as claimed in any preceding claim, wherein the delivery means of a node delivers data messages only when at least one data message received on each of the input channels of said node is stored in the storage means.

8. A complex computing system as claimed in Claim 6 or 7, wherein each node is adapted to perform at least one channel flushing routine triggerable by lack or paucity of channel traffic.

9. A complex computing system as claimed in any preceding claim, wherein all data messages caused by a first data message anywhere in the system are delivered to a node before any messages caused by a second data message, junior to the first data message, are delivered to said node.

10. A method of ordering data messages within a complex computing system comprising a plurality of nodes connected to each other by channels along which data messages are sent and received, the method comprising, for each node, timestamping each message on arrival, queuing messages until a message has been received on each input channel to the node, and delivering the queued messages for processing sequentially in accordance with their timestamps, the message having the most senior timestamp being delivered first, wherein the timestamping at each node is cumulative so that the timestamp of a particular message indicates the seniority acquired by that message, generation by generation, and wherein the seniority of one message against another is determined by the progressive comparison of respective generations in the timestamps until the first distinction exists.

11. A complex computing system comprising a plurality of nodes between which data messages are exchanged, wherein after the arrival at a node of a message, delivery of the message by the node is delayed until after the delivery and consequences of all more senior messages which affect the node.

12. A complex computing system as claimed in any one of Claims 1 to 9 or 11, wherein the system is either a distributed computing system, a symmetric multi-processor computer, or a massively parallel processor computer.

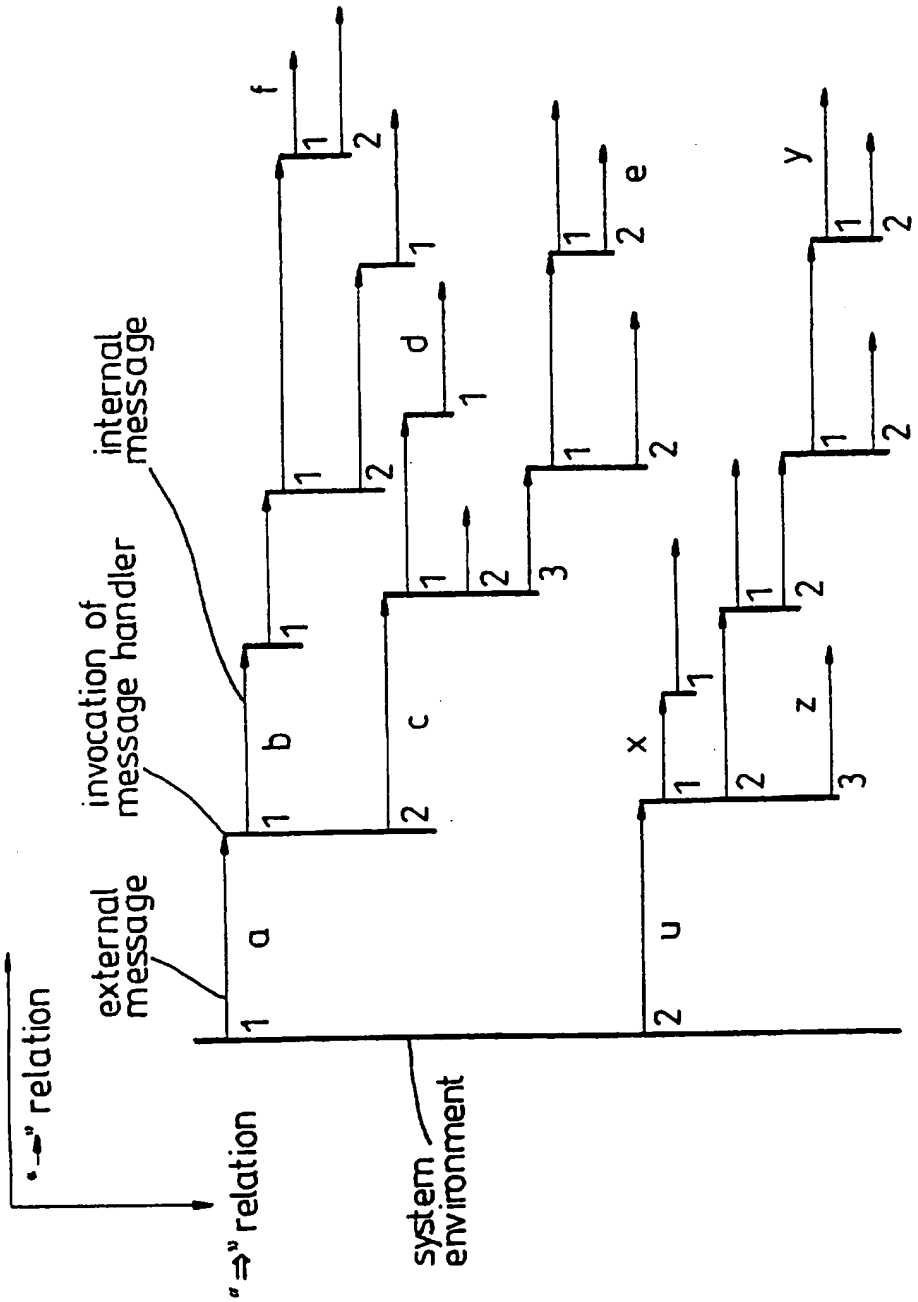

internal message
invocation of message handler
external message
"-" relation
"⇒" relation
system environment
a
b
c
d
e
f
u
x
y
z

Fig. 1

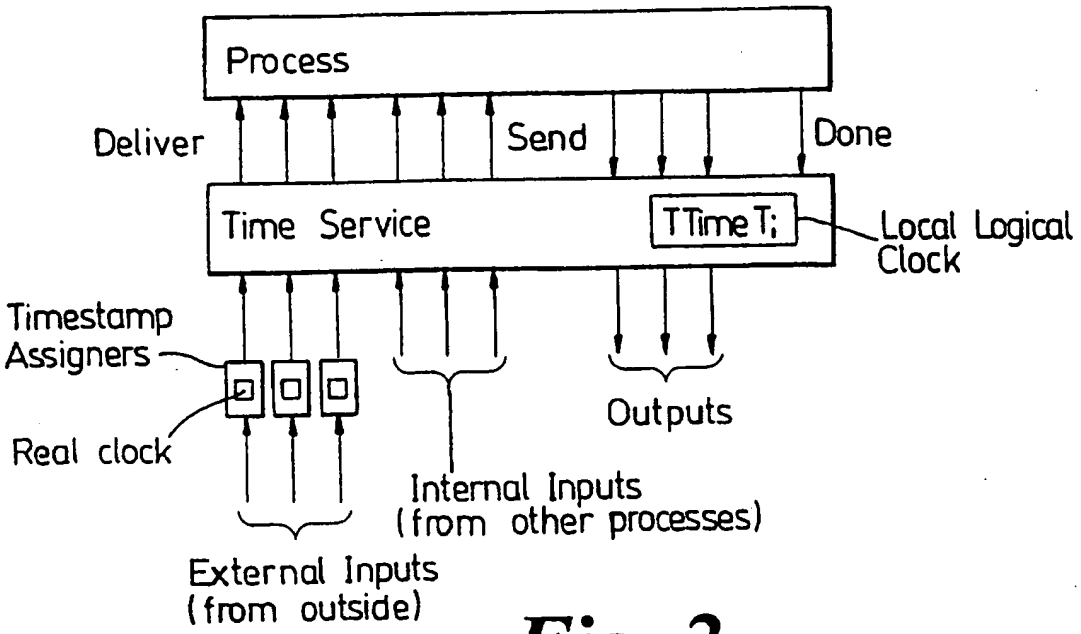

Process
Deliver
Send
Done
Time Service
TTimeT;
Local Logical Clock
Timestamp Assigners
Real clock
Outputs
Internal Inputs (from other processes)
External Inputs (from outside)

Fig. 2

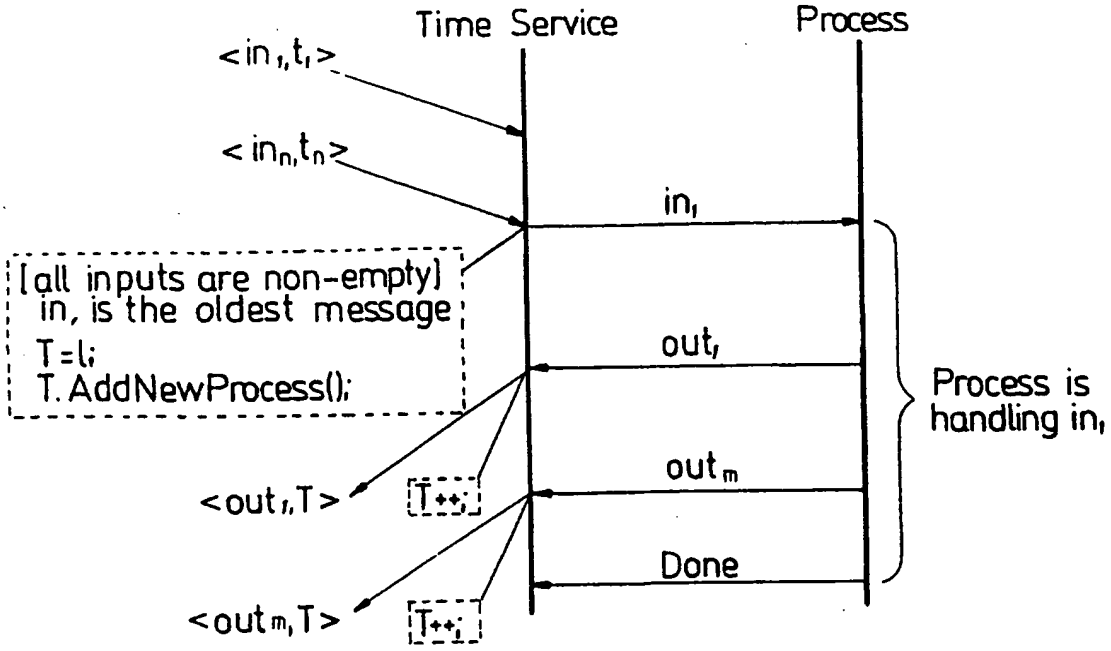

Time Service
Process
<in1,t1>
<inn,tn>
in1
[all inputs are non-empty] in1 is the oldest message
T=t;
T.AddNewProcess();
out1
outm
Done
Process is handling in1
<out1,T>
T++;
<outm,T>
T++;

Fig. 3

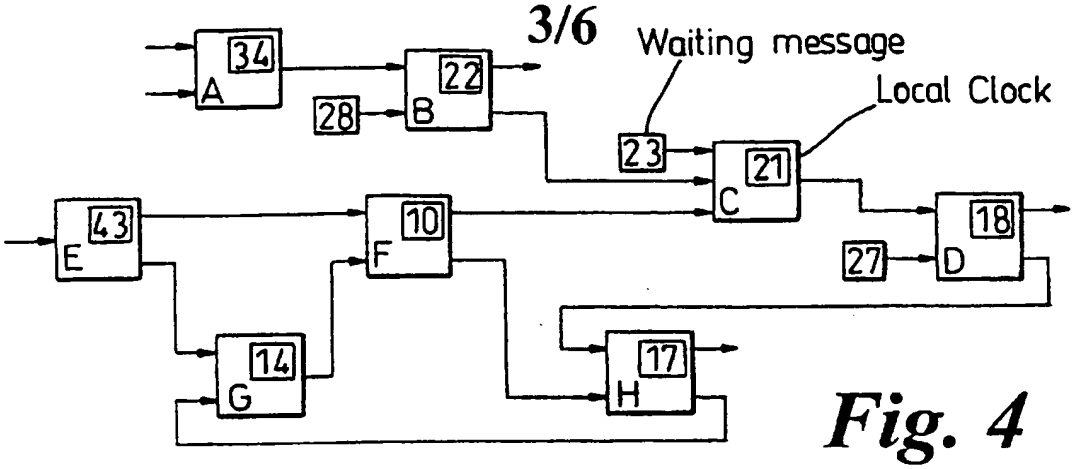

34
A
28
22
B
Waiting message
Local Clock
23
21
C
43
E
10
F
18
27
D
14
G
17
H

Fig. 4

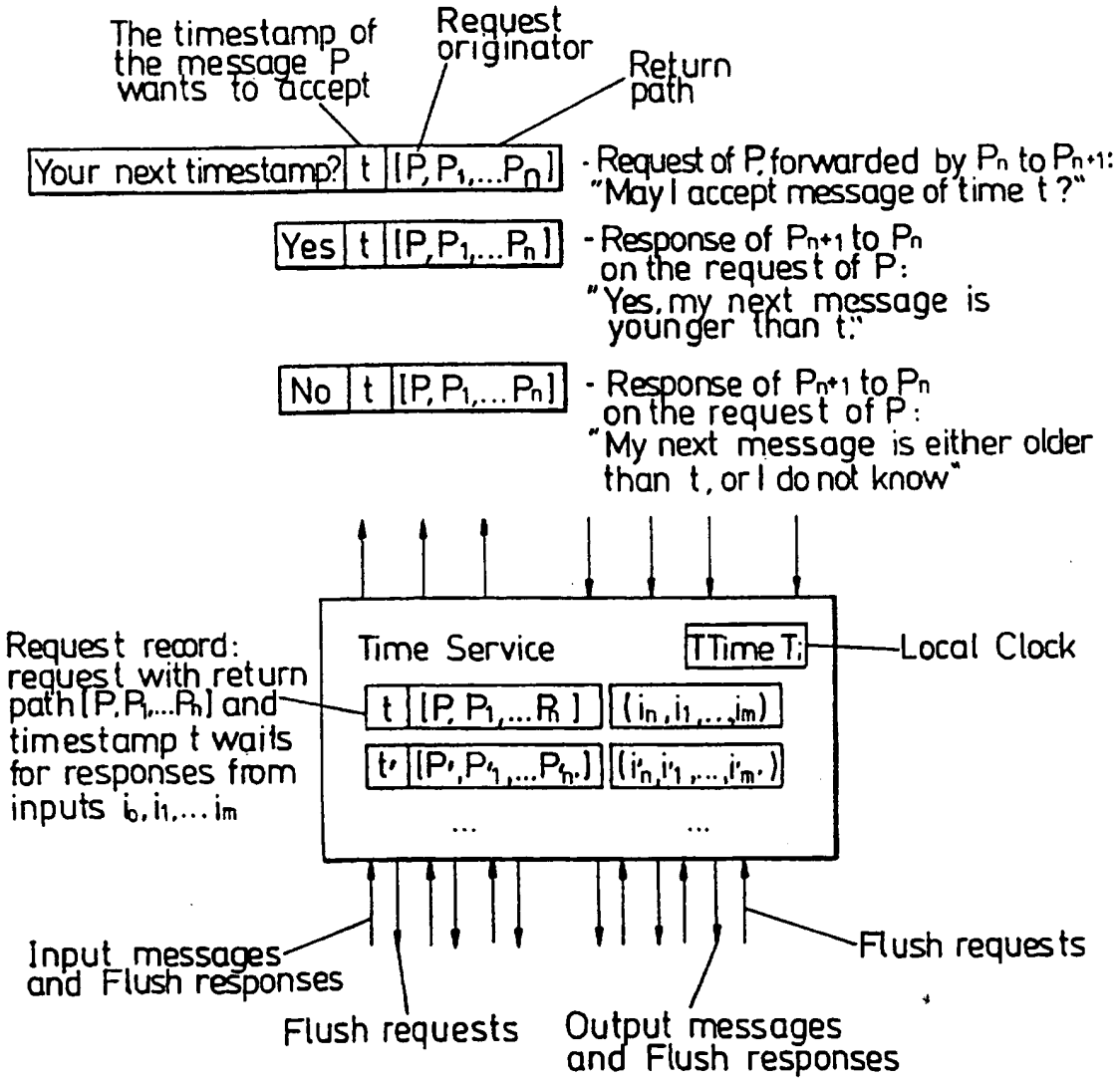

The timestamp of the message P wants to accept
Request originator
Return path
Your next timestamp? | t | [P, P1,... Pn]
- Request of P forwarded by Pn to Pn+1: "May I accept message of time t ?"
Yes | t | [P, P1,... Pn]
- Response of Pn+1 to Pn on the request of P: "Yes, my next message is younger than t."
No | t | [P, P1,... Pn]
- Response of Pn+1 to Pn on the request of P: "My next message is either older than t, or I do not know"
Request record: request with return path [P, P1,... Pn] and timestamp t waits for responses from inputs i0, i1,... im
Time Service
TTime T;
Local Clock
t | [P, P1,... Pn] | (in, i1,..., im)
t' | [P', P'1,... P'n'] | (i'n, i'1,..., i'm')
...
...
Input messages and Flush responses
Flush requests
Output messages and Flush responses
Flush requests

Fig. 5

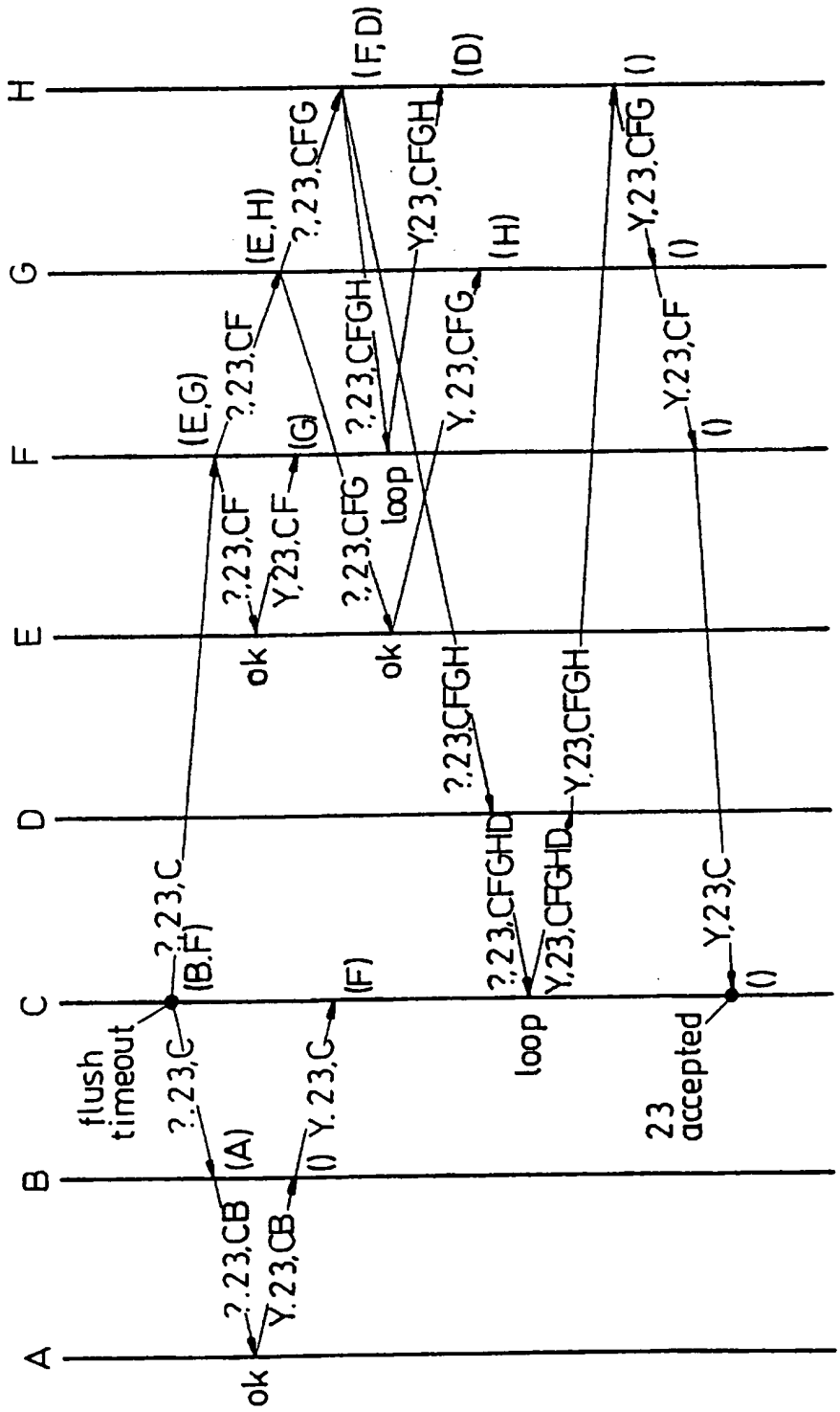

H
G
F
E
D
C
B
A
(F,D)
(D)
(E,H)
?,23,CFG
Y23,CFGH
(H)
Y,23,CFG
(E,G)
?,23,CF
(G)
?,23,CFGH
Y, 23,CFG
Y,23,CF
loop
?,23,C
(B,F)
(F)
?,23,CFGHD
Y,23,CFGHD
Y,23,C
flush timeout
?,23,C
(A)
(!) Y, 23,C
loop
23 accepted
?,23,CB
Y,23,CB
ok
ok
ok
(!)
(!)
(!)

Fig. 6

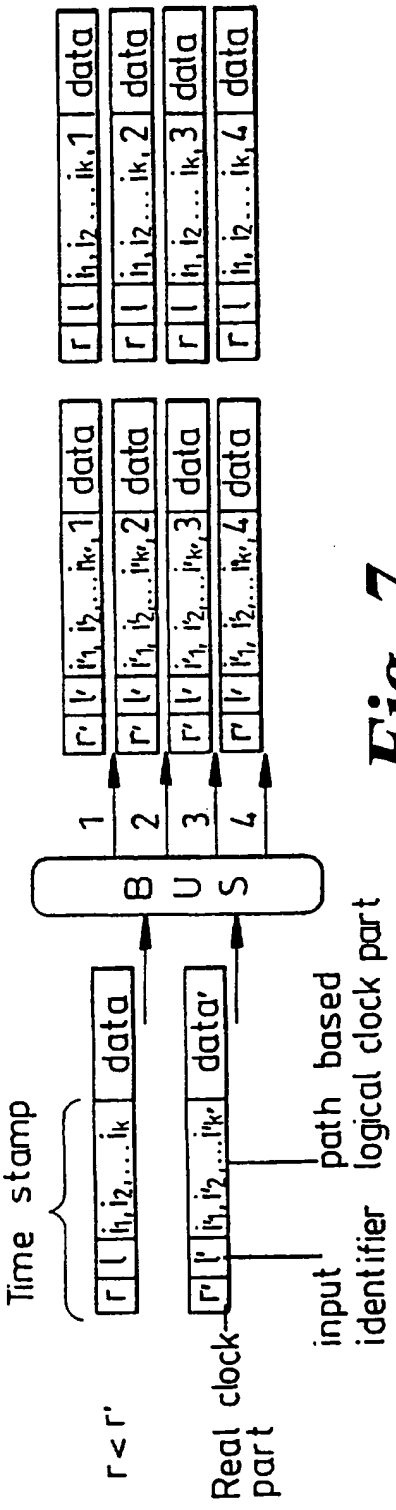


Fig. 7

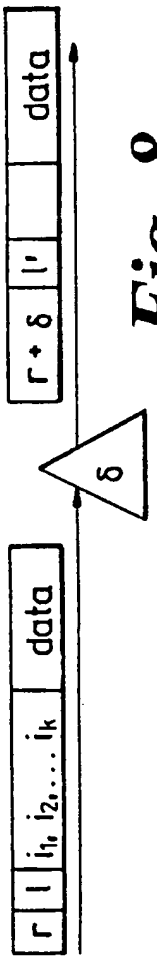


Fig. 8

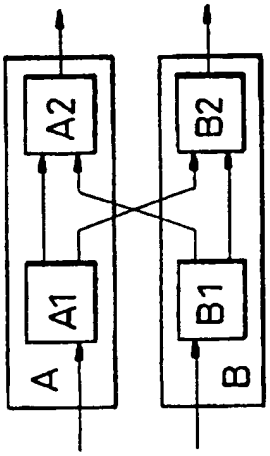


Fig. 9

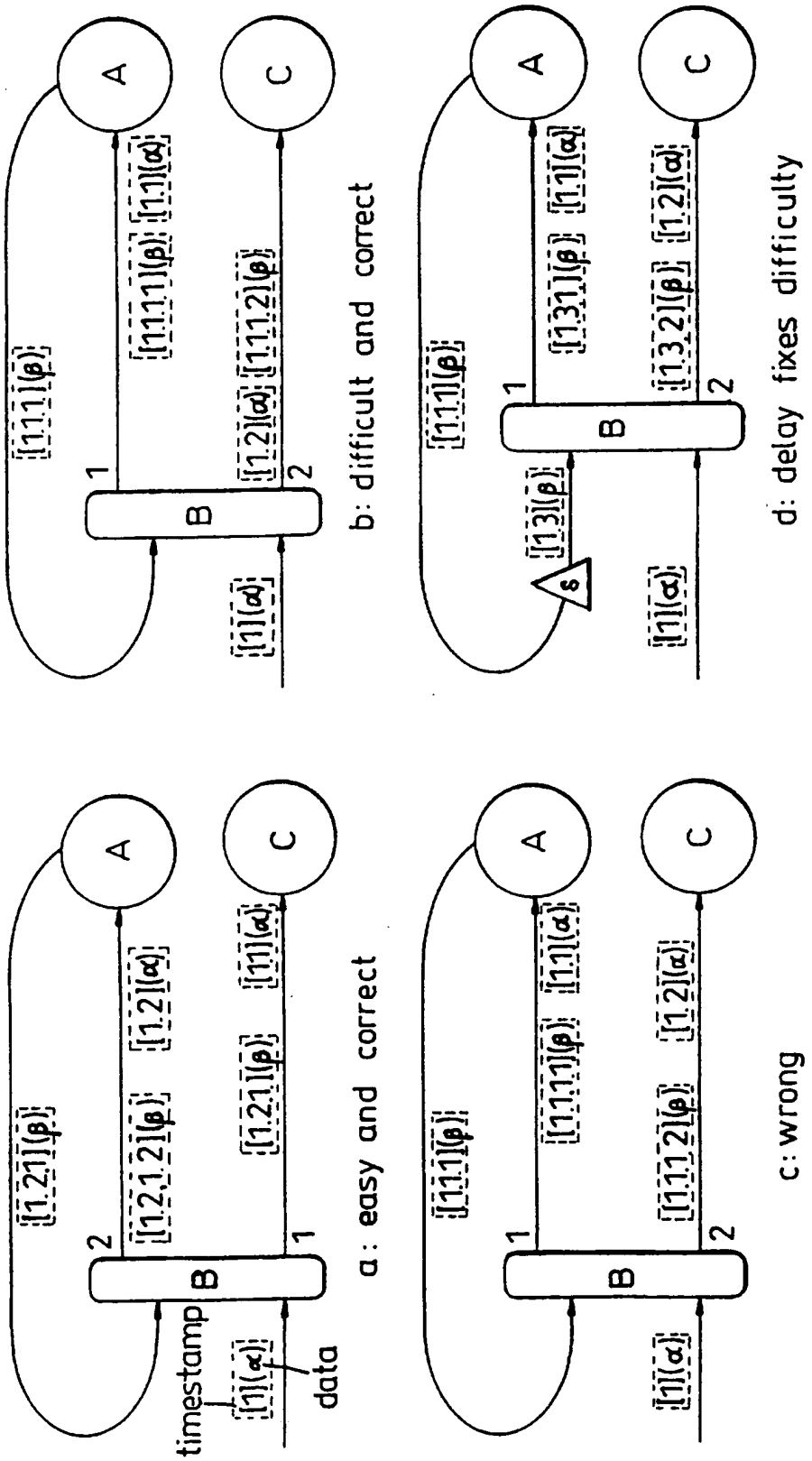

A
B
C
1
2
timestamp
data
[1.2.1](β)
[1.2.1.2](β)
[1.2](α)
[1.2.1](β)
[1](α)
a: easy and correct
[1.1.1](β)
[1.1.1](β)
[1.1](α)
[1.1.1.2](β)
[1.2](α)
c: wrong
[1.1.1](β)
[1.1](α)
[1.1.1](β)
[1.2](α)
[1.1.2](β)
[1.2](α)
b: difficult and correct
[1.1.1](β)
[1.3.1](β)
[1.3.2](β)
[1.2](α)
[1](α)
δ
d: delay fixes difficulty

Fig. 10

INTERNATIONAL SEARCH REPORT

Interna .al Application No
PCT/GB 97/02006

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	JEFFERSON D R: "VIRTUAL TIME" ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, vol. 7, no. 3, July 1985, pages 404-425, XP000614859 cited in the application see the whole document ---	1-12
A	LAMPORT L: "TIME, CLOCKS, AND THE ORDERING OF EVENTS IN A DISTRIBUTED SYSTEM" COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY, vol. 21, no. 7, July 1978, pages 558-565, XP000615783 cited in the application see the whole document ---	1-12

-/--

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
23 October 1997	12. 11. 97
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Fonderson, A

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Interna al Application No
PCT/GB 97/02006

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	KEARNS P ET AL: "AN IMPLEMENTATION OF FLUSH CHANNELS BASED ON A VERIFICATION METHODOLOGY" PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTIN SYSTEMS, YOKOHAMA, JUNE 9 - 12, 1992, no. CONF. 12, 9 June 1992, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 336-343, XP000341030 see the whole document -----	1,8,10, 12

1

Form PCT/ISA/210 (continuation of second sheet) (July 1992)